Receiver-initiated Message Passing over RDMA Networks

Scott Pakin
pakin@lanl.gov
CCS-1 Performance and Architecture Lab (PAL)
Los Alamos National Laboratory

Abstract

Providing point-to-point messaging-passing semantics atop Put/Get hardware traditionally involves implementing a protocol comprising three network latencies. In this paper, we analyze the performance of an alternative implementation approach—receiver-initiated message passing—*that eliminates one of the three network latencies. Performance measurements taken on the Cell Broadband Engine indicate that receiver-initiated message passing exhibits substantially lower latency than standard, sender-initiated message passing.*

1. Introduction

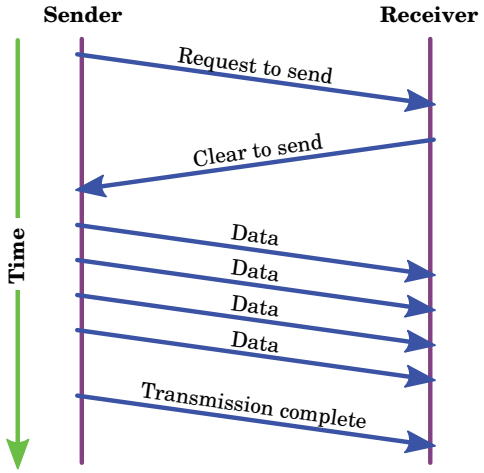
Remote Direct Memory Access (RDMA)—also known as the PUT/GET or one-sided communication model—is a mechanism for transferring data over a network to (PUT) or from (GET) an address in another node's memory without the explicit involvement of the remote node's CPU. RDMA is supported by all of the major cluster interconnects, including InfiniBand [10], QsNet^{II} [1], iWARP-enabled Ethernet [9], and Myrinet GM [18]. The advantages of RDMA are that it reduces memory copies, decouples the CPU from data transfer, and relaxes data-ordering constraints, all of which can help improve raw communication performance.

Few programmers, however, write explicit PUT/GET code. Instead, it is far more common in the context of clusters and parallel computers to program using the MPI [22] interface, the de facto standard for message passing (a.k.a. the two-sided communication model). MPI provides a rich set of communication primitives and is portable to a wide variety of network architectures, including those based on RDMA communication.

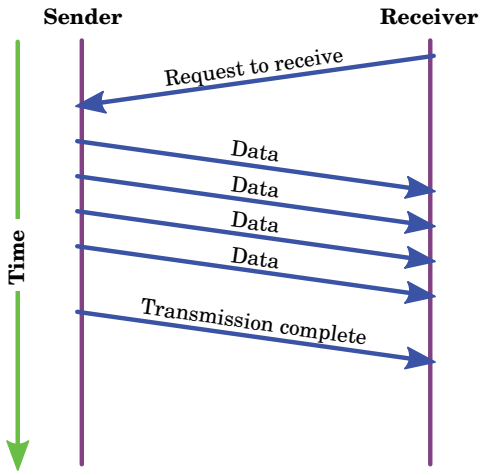
This paper presents and evaluates the performance of a mechanism for implementing MPI-style message passing atop RDMA networks. While our approach is applicable to RDMA networks in general—and can easily be extended even to traditional send/receive networks—our initial implementation targets networks of Cell Broadband Engine (CBE or simply "Cell") processors [5]. A Cell contains a single 64-bit PowerPC processor (the PPE) and eight vector processors called *synergistic processing elements* (SPEs). The Cell is an interesting target for a high-performance messaging layer because of its high peak performance: 204.8 Gflop/s (billion floating-point operations per second) in single precision or 14.6 Gflop/s in double precision. Because each SPE can directly access only 256 KiB[1] of memory—on-chip SRAM known as *local store*—a typical application must communicate frequently to stage data held in off-chip main memory into and out of local store and to coordinate the SPEs. Data transfers are performed primarily using explicit DMA operations (PUT and GET) while coordination and synchronization can exploit a variety of hardware mechanisms [13]. Because communication must be fairly fast in order to keep the vector units busy it is worthwhile to optimize the performance of the software communication layer. In this paper we describe how we managed to provide send/receive performance that rivals the lower-level PUT/GET performance and significantly improves upon the performance of previously published results.

The remainder of this paper is structured as follows. Section 2 presents the key concept underlying our approach—receiver-initiated message passing—and describes the implementation in a high-level, network-independent manner. We then contrast our work to other projects in Section 3. Cell-specific implementation details are provided in Section 4. In Section 5 we analyze the performance of our messaging layer and discuss its performance relative to the hardware performance and relative to comparable messaging layers. Finally, we draw some conclusions from our work in Section 6.

[1] This paper follows the IEC convention [23] that KB, MB, and GB represent 10^3, 10^6, and 10^9 bytes while KiB, MiB, and GiB represent 2^{10}, 2^{20}, and 2^{30} bytes.



(a) Traditional, sender-initiated message passing



(b) Our receiver-initiated message-passing technique

Figure 1: Sender- vs. receiver-initiated message passing

2. Receiver-Initiated Message Passing

Message passing over RDMA is traditionally implemented using a sender-initiated protocol as shown in Figure 1(a). First, the sender transmits a control message to the receiver, specifying the number of bytes it intends to transmit. Next, the receiver locates or allocates resources (e.g., a buffer in which to store the message) and transmits a control message back to the sender indicating the address to which the data should be written. Finally, the sender uses a PUT mechanism to transmit the data and a completion flag to notify the receiver that the PUT has completed. (In some implementations, the receiver does a GET of the data followed by transmitting a control message to indicate that the GET has completed.) Similar procedures are used for send/receive-based message passing.

The key innovation behind our messaging layer's implementation is the use of *receiver-initiated* message passing instead of the usual sender-initiated message passing. Figure 1(b) illustrates this protocol. First, the receiver allocates resources and transmits a control message back to the sender indicating the buffer size and the address to which the data should be written. Then, the sender uses a PUT mechanism to transmit the data and a completion flag to notify the receiver that the PUT has completed. As should be evident from Figure 1, receiver-initiated message passing pays two network latencies instead of the usual three. This reduction in network latencies is intended to produce lower-latency, higher-bandwidth communication than is possible with sender-initiated message passing.

The importance of receiver-initiated message passing is not so much a reduction in the total number of messages but a reduction in the number of synchronization points—the number of times that one process must wait for a message from another process to traverse the network. That is, Figure 1(b) represents receiver-initiated message passing regardless of whether the data and completion flag are sent with a single PUT or any number of back-to-back PUTs. Synchronizations expose the network latency, and receiver-initiated message passing transmits data with fewer synchronizations than are needed by sender-initiated message passing.

So as to avoid any confusion we should state that receiver-initiated message passing is not equivalent to messaging based on RDMA GET. Receiver-initiated message passing can conceivably be based on either GET or PUT—our initial implementation (Section 4) uses PUT—while GET-based communication can be either sender- or receiver-initiated. As a concrete example, consider Karamcheti and Chien's "pull-based" messaging [12], one of the earliest attempts to achieve robust, high-performance communication on a system providing hardware support for remote GETs and PUTs. While the data transfer proper is in fact initiated by the receiver in Karamcheti and Chien's approach, the messaging protocol *as a whole* is initiated by the sender. (The sender enqueues a send request on a receive-side queue; the receiver GETs the data from the sender; then, the receiver notifies the sender that the buffer can be reused.) Consequently, we consider pull-based messaging and its intellectual offspring as sender-initiated message passing, not receiver-initiated message passing.

Given that receiver-initiated message passing is such a simple modification to sender-initiated message passing, why is it not more commonly utilized? We suspect that many researchers have considered receiver-initiated message passing but rejected its use because of one fundamental limitation: A receiver needs to know which sender will send to it. Consequently, MPI's

`MPI_ANY_SOURCE` specifier [22], a wildcard-receive mechanism, is nontrivial to implement efficiently in the context of receiver-initiated message passing. Our perspective is that (1) small-memory processors such as the Cell SPE cannot effectively run a complete MPI implementation to begin with; and, (2) most applications that use MPI—especially those that are likely to be ported to networks of Cells—utilize sufficiently regular communication patterns that they do not inherently rely upon wildcard receives. If we are correct in these assumptions then it makes sense to sacrifice support for `MPI_ANY_SOURCE` to gain improved communication performance for the common case, in which a receiver knows *a priori* which senders will give it data.

3. Related Work

Receiver-initiated communication has previously been used in the contexts of multicast communication [24], load sharing [6], and media-access control protocols [25]. Our work represents one of the only implementations of receiver-initiated communication in the context of point-to-point communication in high-performance messaging layers for parallel applications. The Fast Communication Interface (FCI) [3], a low-level messaging layer designed as part of the Swiss-T*x* project, is one of the only published examples of receiver-initiated message passing. FCI even implements wildcard receives. However, the implementation of wildcard receives requires that the sender acquire a global lock on the receive request before transferring any data. While a per-message global lock is unlikely to cause severe performance degradation on a small cluster—the final Swiss-T1 system contained only 64 processors [8]—it is likely to be unusably slow on the cluster that our work is targeting: The Roadrunner system soon to be installed at Los Alamos National Laboratory will contain over 100,000 SPE processor cores. As we stated in Section 2, it is likely that others have considered but dismissed the use of receiver-initiated message passing on the grounds that it cannot efficiently implement wildcard receives. Our argument is that the gain in communication performance outweighs this minor limitation.

In terms of providing an MPI [22] implementation for the Cell SPEs, the most closely related projects to ours are the Cell MPI implementations from Kumar, Krishna, et al. [15–17]. The key philosophical difference between our work and theirs is that we assume that the data buffers to be communicated reside in the SPEs' local stores while Kumar, Krishna, et al. assume that these buffers reside in main memory. It is too early to tell which assumption presents application developers with a more convenient programming model. However, communication across local stores can exploit an aggregate communication bandwidth of 204.8 GB/s (190.7 GiB/s) while communication between main-memory buffers requires two crossings (source memory to local store and local store to destination memory) of a shared 25.6 GB/s (23.8 GiB/s) channel [13]. For messages that fit in the small (256 KiB) SPE local store, our approach therefore has the potential for an eight-fold performance improvement over Kumar, Krishna, et al.'s MPI implementations based solely on the available aggregate bandwidths. Beside the philosophical difference, we show in Section 5.3 that our receiver-initiated message-passing mechanism enables our MPI implementation to provide 30.5% better latency than Kumar, Krishna, et al.'s in an apples-to-apples comparison.

Ohara et al. [20] have implemented an MPI-like programming model for the Cell based on the notion of a *microtask*, a unit of computation and its associated data that fit in a SPE's local store—essentially a virtual SPE. A scheduler decides where and when to run each microtask. Although Ohara et al.'s microtask communication is based on MPI, this model does assume finer-grained units of work than are used by a typical MPI program. An advantage of the microtask model relative to the traditional coarse-grained MPI model is that microtasks can be scheduled easily and flexibly. A disadvantage is that microtask-scheduling overhead may turn out to incur a significant performance cost relative to a statically scheduled MPI program.

High-performance messaging-layer implementations boomed in the early-to-mid 1990s with such projects as Active Messages [27], Fast Messages [21], U-Net [26], and VMMC [2]. With the emergence of MPI as the dominant interface for parallel programming, much messaging-layer research now focuses on improving communication performance in the context of MPI. A recent example is Buntinas et al.'s Nemesis messaging layer [4]. Like our messaging layer, Nemesis uses high-speed but nonscalable mechanisms when communicating within a node and lower-speed but more scalable mechanisms when communicating outside a node. However, Nemesis's high-speed mechanisms such as its "fastbox" and shadow head pointer assume cache-coherent shared memory while our high-speed mechanisms assume distributed memory and explicit RDMA-based data transfers. Also, because Nemesis is intended to run on conventional microprocessors with abundant memory it is able to support the complete MPI specification while our messaging layer provides only a subset so as to fit in the limited Cell local store. Because of these differences a performance comparison between Nemesis and our messaging layer is of only limited value. Nevertheless, to make the raw performance data explicit, Nemesis observes 0.341 µs latency and 1.5 GiB/s bandwidth between the

two cores of a dual-core 2.0 GHz Opteron socket [4] versus our 0.272 µs latency and 22.9 GiB/s bandwidth between two SPEs of an 8-SPE 3.2 GHz Cell socket.

4. Implementation

Our initial implementation of receiver-initiated message passing targets clusters of Cell processors. We rather generically call this initial implementation the *Cell Messaging Layer*. The programming model underlying the Cell Messaging Layer is that applications run entirely on the SPEs with one rank in the computation per SPE. The Cell Messaging Layer treats the PPE as an intelligent network interface card (NIC) and uses it solely to communicate with other Cells.

The Cell Messaging Layer's application programming interface (API) provides a subset of MPI's functions and semantics [22]. Section 4.4 lists what the Cell Messaging Layer does and does not implement. Briefly, its limitations stem primarily from the extremely limited amount of local store available to each SPE. The Cell Messaging Layer's entire footprint in local store is 10,768 bytes (8,624 code + 2,144 data). This is for a build that supports clusters of symmetric multiprocessors (SMPs) containing up to 16 SPEs per SMP node. Each additional SPE within a node requires approximately 64 bytes of additional local store. No additional local store is required to communicate with a SPE in a different node because this communication is handed off to the PPE, which performs the internode communication using exclusively main memory. The remainder of Section 4 explains in detail the Cell Messaging Layer's data structures and communication protocols.

4.1. Cell overview and primitive performance

Figure 2 presents a high-level illustration of the Cell processor. The important architectural characteristics from the standpoint of a messaging-layer implemention are the following:

- Each of the eight Synergistic Processor Elements (SPEs) contains a 3.2 GHz, in-order Synergistic Processor Unit (SPU) vector processor plus only 256 KiB of directly addressible high-speed memory (local store). Communication with other SPEs and access to main memory must be perfomed via DMA operations.
- The PowerPC Processor Element (PPE) contains a 3.2 GHz, 64-bit, in-order PowerPC Processor Unit (PPU). In our experience, the PPE is a fairly slow processor—a third as fast as a 2.2 GHz Opteron or a 1.9 GHz Power5 on a small kernel application we benchmarked. Hence, the PPE is better utilized as a communication processor than as a compute processor.
- The SPEs, PPE, memory interface controller (MIC), and broadband interface (BIF)—the connection to other Cells within an SMP—are interconnected via a high-speed Element Interconnect Bus (EIB). The EIB comprises four rings—two in each direction—and a central arbiter. In the absence of path contention, each ring can perform three concurrent data transfers. The SPEs, PPE, and MIC each have 25.6 GB/s (23.8 GiB/s) links to and from the EIB. The BIF bandwidth is configurable at boot time. The combined bandwidth for the BIF and (not shown in Figure 2) system I/O is 25.0 GB/s inbound and 35.0 GB/s outbound. On our test platform the BIF is configured with 20.0 GB/s (18.6 GiB/s) inbound and 30.0 GB/s (27.9 GiB/s) outbound, or effectively 20.0 GB/s in each direction for the two directly connected Cells.

The Cell provides a number of synchronization mechanisms for coordinating the PPE and the SPEs. Mailboxes are a convenient way to transmit a small number of 32-bit units of data. However, mailbox performance is surprisingly poor: We measured a half round-trip time (½ RTT) of 5-6 µs for mailbox communication between a SPE and the PPE! The fetch-and-add mechanism is useful for implementing wait-free data structures—in particular, message queues. We measured the Cell's fetch-and-add latency (via the SPE library's `atomic_add_return()` function) as 0.134 µs. However, because fetch-and-add is an inherently serial operation, its latency increases linearly with contention. With all eight SPEs performing simultaneous fetch-and-add operations, one would expect to observe up to $8 \times 0.134 = 1.072$ µs latency, which we consider too high for intra-socket communication.

Because of the high costs of the various synchronization primitives we decided to structure the Cell Messaging Layer to use exclusively DMA-based synchronization (e.g., waiting for a flag to be set by a DMA PUT). Furthermore, because all communication between the SPEs and the PPE is serialized at the PPE we decided to structure the Cell Messaging Layer to avoid the PPE for all intra-Cell communication.

4.2. Point-to-point communication

Intra-Cell communication Figure 3 illustrates the data transfers used in SPE-to-SPE communication. SPE-to-SPE communication follows the basic receiver-initiated message passing approach outlined in Section 2.

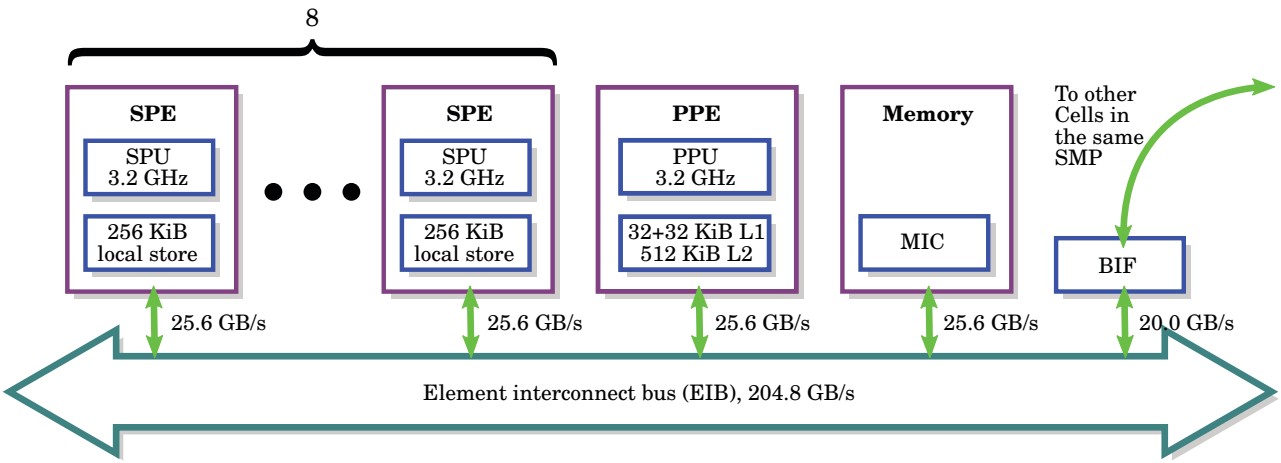


Figure 2: Block diagram of the Cell processor

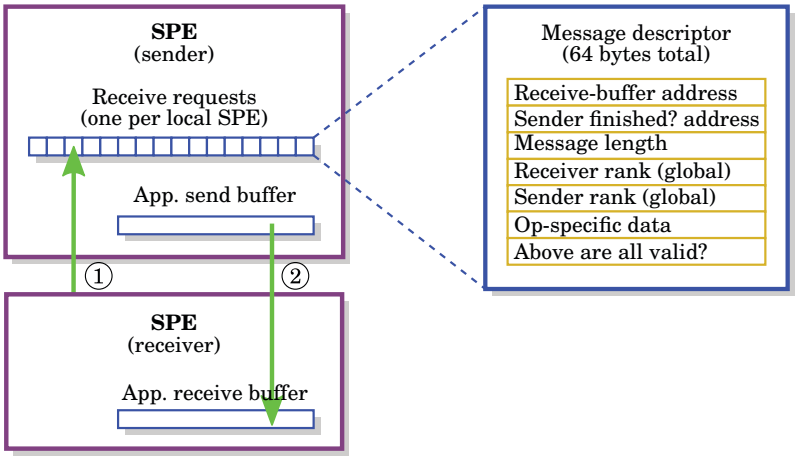


Figure 3: Intra-Cell communication in the Cell Messaging Layer

In Step 1, the receiver PUTs a message descriptor into the sender's receive-request slot associated with the receiver's local rank. In fact, this action comprises two PUT operations. The first PUT transfers the address of the application's receive buffer, the address of the sender's *Sender finished?* flag, message length, global ranks of the sender and receiver, and any operation-specific data (described in Section 4.3). The second PUT—which is "fenced" to force the first PUT to complete first—specifies that the first PUT's data is ready to be read. (Recall from the discussion in Section 2 that the use of multiple back-to-back PUTs does not negate the fact that this communication protocol is an instance of receiver-initiated message passing.) In Step 2, when the sender observes the receive request, it PUTs data directly from the application's send buffer into the application's receive buffer on the receiver SPE followed by a fenced PUT to the *Sender finished?* flag address. No explicit copies are made of the data on either the send side or the receive side.

Inter-Cell communication Because we are interested in clusters of Cells, not just individual Cell processors, we implemented a mechanism for forwarding data from a SPE to its PPE then across a network to a remote PPE and back down to the target SPE. The PPE needs to be involved because a SPE cannot interact directly with I/O-bus devices such as NICs. Rather than write a PPE-to-PPE messaging layer from scratch we chose to rely on an existing MPI implementation. (Any MPI that runs on the PowerPC should run unmodified on the PPE.) Originally, we used receiver-initiated message passing for the entire end-to-end transfer. However, doing so produced poor performance because the receive request and the data transfer each require an MPI message, and each MPI message may internally require multiple network

crossings for MPI to exercise flow control on the data. We therefore restructured the code to use sender-initiated message passing at the PPE-to-PPE level to better exploit the existing MPI implementation.

Our revised data path is illustrated in Figure 4. Communication between the SPEs and the PPE is achieved through a circular queue of send requests, a circular queue of receive requests, and atomic operations on each queue's free-slot counter and free-slot index. A brief summary of the protocol is as follows: The source SPE enqueues the message data and a send request on its PPE's send queue while the target SPE enqueues a receive request on its PPE's receive queue; the send-side PPE transfers the message to the receive-side PPE using MPI; and finally, the receive-side PPE transfers the data to the target SPE and notifies it when the data is ready.

We now examine this protocol in more detail, referring to the numbered steps in Figure 4. In Step 1 on the send side and Step 5 on the receive side—these steps can occur in parallel—the SPE repeatedly performs an `atomic_dec_if_positive()` operation on the free-slot counter until it receives a nonnegative number, implying that the queue contains a free send/receive request slot. Next, in Step 2 on the send side and Step 6 on the receive side, the SPE performs an atomic increment of the free-slot index and takes the result modulo the queue length to determine which queue slot to write to. In Step 3, the sender PUTs the message data into the appropriate slot in the send queue then performs a fenced PUT to the *valid?* flag. Analogously, in Step 7 the receiver PUTs the message data into the appropriate slot in the receive queue then performs a fenced PUT to the *valid?* flag. The send-side PPE invokes `MPI_Isend()` to transmit the data through the network to the remote PPE in Step 4, and the receive-side PPE invokes the corresponding `MPI_Irecv()` in Step 8. When the data arrives at the receiver, the PPE transfers that data to the SPE either with a simple `memcpy()` into the SPE's local store or by instructing the SPE to GET the data from main memory and signal the PPE when the GET completes by setting a flag in local store. We determined empirically that the `memcpy()` approach is faster for messages of length 128 bytes or less because of the low startup overhead and that the GET approach is faster for longer messages because of the higher memory bandwidth available to DMA operations.

Inter-Cell communication, besides being practical on its own, also demonstrates that receiver-initiated message passing can coexist with conventional, sender-initiated message passing. It may thereby be possible to efficiently implement wildcard receives in an MPI implementation that uses receiver-initiated message passing by falling back onto a lower-performing sender-initiated scheme when `MPI_ANY_SOURCE` is needed. Constructing such a scheme is beyond the scope of this paper, but a nonscalable approach is discussed by Brauss et al. [3].

As an aside, the Cell Messaging Layer additionally uses the SPE-to-PPE and PPE-to-SPE paths to implement a remote-procedure call (RPC) mechanism that enables a SPE to invoke functions on the PPE and receive the results. We have found this to be convenient for enabling SPEs to dynamically allocate main memory, for example.

4.3. Collective communication

Given point-to-point messaging passing it is straightforward to implement collective operations. In the Cell Messaging Layer's barrier implementation, each SPE notifies its PPE that it has entered the barrier, then all SPEs synchronize with each of their immediate neighbors (within the same Cell) on a binary hypercube. Concurrently, all of the PPEs perform an `MPI_Barrier()`. Finally, each PPE notifies its local SPEs that they can exit the barrier.

Broadcasts and reductions are implemented using binomial trees, which have long been known to be an efficient mechanism for such operations [11]. Binomial trees are particularly useful for clusters such as ours (Section 5) in which each node is in fact an SMP containing *two* Cells. Although the hardware supports seamless RDMA operations between the two Cells—they can appear as a single Cell with double the number of SPEs—inter-Cell data transfers observe significantly less bandwidth than intra-Cell data transfers (about a fifth according to our measurements). One advantage of a binomial tree is that only a single message traverses the slow, inter-Cell link in the common case of local rank 0 serving as the root of the broadcast/reduction. All other communication traverses only the fast, intra-Cell EIB links.

Broadcasts and reductions are implemented hierarchically, separating local (intra-Cell) processing from global (MPI) processing. Internally, when invoking point-to-point operations, the Cell Messaging Layer's broadcasts and reductions utilize the operation-specific data field shown in Figures 3 and 4 to transmit metadata such as the root rank, the reduction function (e.g., `MPI_SUM`), and the reduction datatype (e.g., `MPI_DOUBLE`).

4.4. Current limitations

The Cell Messaging Layer does not provide a complete MPI implementation. At the time of this writing it includes only the following functions:

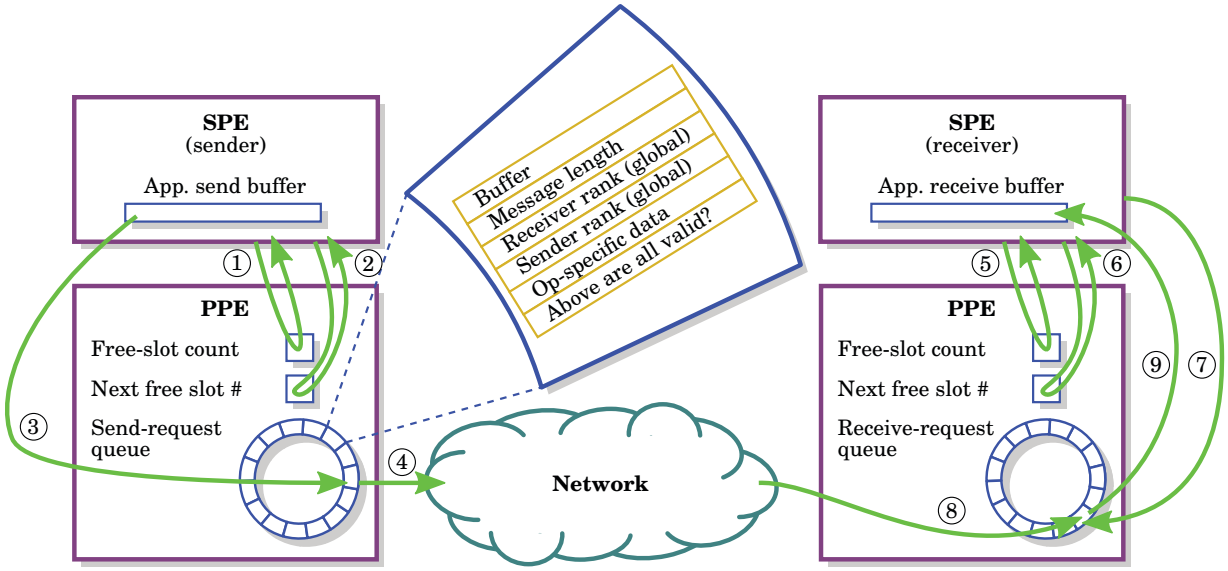


Figure 4: Inter-Cell communication in the Cell Messaging Layer

`MPI_Init()`, `MPI_Comm_rank()`, `MPI_Comm_size()`, `MPI_Send()`, `MPI_Recv()`, `MPI_Barrier()`, `MPI_Bcast()`, `MPI_Reduce()`, `MPI_Allreduce()`, `MPI_Wtime()`, `MPI_Abort()`, and `MPI_Finalize()`. The Cell Messaging Layer was developed in conjunction with an effort by Los Alamos National Laboratory's Performance and Architecture Lab to port the Sweep3D neutron-transport kernel [14] to the Cell processor, and the preceding functions are sufficient for implementing Sweep3D's communication routines.

In addition to the Cell Messaging Layer's limited function set, `MPI_COMM_WORLD` (the set of all SPEs) is the only supported communicator; there are no derived datatypes; message tags are currently ignored; `MPI_Send()` exhibits `MPI_Ssend()`'s semantics of always synchronizing the sender and the receiver; and `MPI_ANY_SOURCE` is not implemented. Subcommunicators, derived datatypes, message tags (including `MPI_ANY_TAG`), a non-synchronizing `MPI_Send()`, and many of the omitted MPI functions (e.g., nonblocking point-to-point operations and additional collective routines) should all be straightforward to implement but can be expected to exact a slight toll in common-case performance. As stated previously, `MPI_ANY_SOURCE` conflicts with the notion of receiver-initiated message passing and may be challenging to implement in an efficient, scalable manner in such a setting.

More functionality will be added to the Cell Messaging Layer in the future; the purpose of our initial implementation is to demonstrate the performance of receiver-initiated message passing and to be able to port some simple applications to the Cell. As more applications are ported, the Cell Messaging Layer will grow to include the functions and semantics needed by those applications.

5. Performance Evaluation

Having argued that receiver-initiated message passing is expected to improve communication performance and described the embodiment of receiver-initiated message passing in the Cell Messaging Layer, we now examine how well the Cell Messaging Layer performs on actual hardware. For completeness we also include performance measurements for inter-Cell communication even though, as Section 4.2 explained, that code path uses traditional, sender-initiated message passing.

The platform on which we perform all of our experiments is a small cluster of Cells. Each node in the cluster contains a QS21 [19] blade. The QS21 organizes two 3.2 GHz CBEs into an SMP configuration. The cluster is interconnected with an InfiniBand [10] network. Each node contains a single InfiniBand 4X SDR NIC. For inter-Cell communication we run the Open MPI [7] implementation of MPI on the PPEs.

5.1. Cell primitive performance

To help relate the Cell Messaging Layer's performance to the performance of the underlying communication primitives, Table 1 presents the results of a set of measurements of various Cell data-transfer primitives. All transfers were performed within a single Cell. All

Initiator	Source	Target	Operation	Throughput (MiB/s)
SPE	Local store	Local store	PUT	24,413
SPE	Local store	Local store	GET	24,413
SPE	Local store	Main memory	PUT	11,321
SPE	Main memory	Local store	GET	6,691
PPE	Main memory	Local store	`memcpy()`	1,213
PPE	Local store	Main memory	`memcpy()`	199

Table 1: Performance of CBE communication primitives

measurements represent the time for 1,000,000 back-to-back operations on units of 16 KiB of data. The `memcpy()` from main memory to local store includes flushing the data from cache. The `memcpy()` from local store to main memory is optimized to copy 16 bytes at a time instead of 8.

Some points to note from the data in Table 1 are that PUT and GET have equal performance—right at the theoretical maximum—when transferring between local stores, but PUTs are significantly faster than GETs when main memory is involved. The Cell Messaging Layer therefore uses exclusively PUTs except when the PPE needs to transfer data to a SPE (Section 4.2). PPE-based `memcpy()` does not exploit the Cell's DMA hardware and therefore observes extremely poor throughput.

5.2. Cell Messaging Layer point-to-point performance

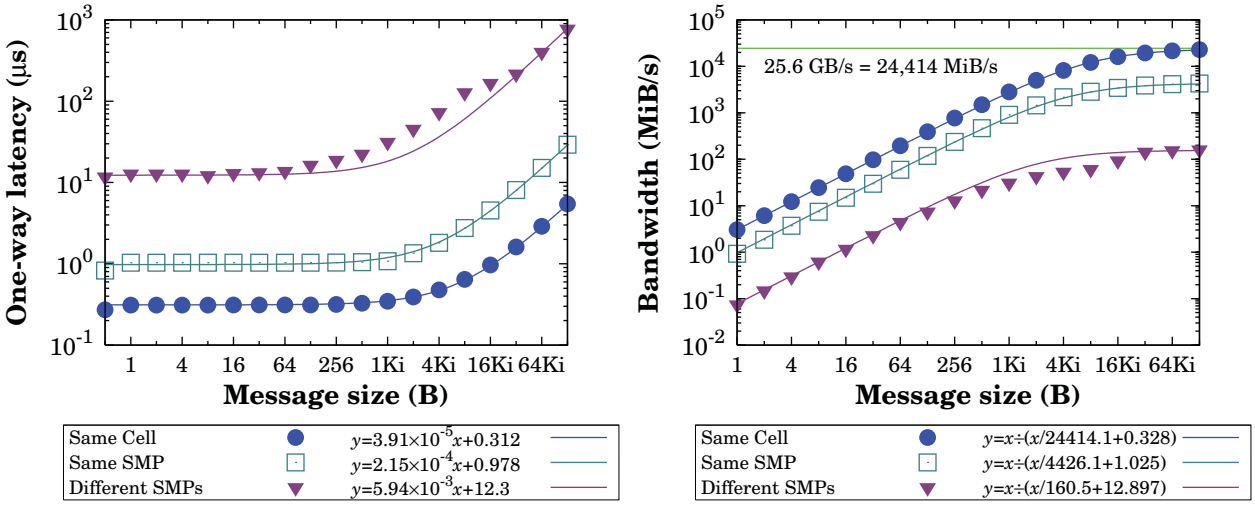
We measure point-to-point communication latency by having one SPE execute `MPI_Send()` followed by `MPI_Recv()` while a second SPE executes the matching `MPI_Recv()` and `MPI_Send()`. This pattern is repeated 10,000 times (with 10 warmup repetitions) and the one-way latency is reported as the total time divided by 20,000—half the average round-trip time (½ RTT). We report bandwidth as the byte count divided by the latency then scaled to units of MiB/s. Figure 5 presents the performance results for three sets of SPEs: "Same Cell", in which the two SPEs communicate directly over the EIB; "Same SMP", in which the two SPEs are on different Cell sockets on the same QS21 node, therefore requiring crossings of both the EIB and BIF; and, "Different SMPs", in which the two SPEs are on different nodes, therefore requiring additional communication through the I/O subsystem and across the InfiniBand network. The graphs in Figure 5 include both measured data (points) and the results of linear regressions on that data (lines). The intra-Cell and intra-SMP curves are quite smooth and are matched well by the regressions. This is to be expected given that the Cell Messaging Layer essentially just sets up a data transfer and lets the Cell's DMA hardware perform it.

The table in Figure 5(c) states each 0 B latency from Figure 5(a) and each 128 KiB bandwidth from Figure 5(b). To put these numbers in perspective, the table further shows each latency figure in terms of equivalent SPE clocks (at 3.2 GHz) and each bandwidth figure in terms of the equivalent single- and double-precision floating-point operation rate. The significance of these values is that they quantify the minimum amount of computation that needs to be performed per floating-point operation to shift the performance bottleneck from communication to computation. For example, a piece of code that is run entirely within a single Cell should perform three double-precision floating-point operations for every five double-precision numbers sent or received via the Cell Messaging Layer (a ratio of 1:0.6).

The not-yet-released eDP (enhanced double precision) version of the Cell increases the peak 8-SPE double-precision performance from 14.6 Gflop/s to 102.4 Gflop/s with single-precision performance remaining at 204.8 Gflop/s. We measured the Cell Messaging Layer on a prototype eDP board at IBM and found similar performance to that presented in this paper. (This is to be expected; only the floating-point pipeline, not the integer pipeline, has so substantially improved.) Consequently, on the eDP the values in the double-precision column of Figure 5(c) are almost exactly half of the values in the single-precision column, which themselves remain unchanged.

According to Figure 5, the Cell Messaging Layer observes a small minimum latency (0.272 µs—a cost equivalent to 870 SPE clock cycles) and a large maximum bandwidth (22,944 MiB/s) in the intra-Cell case.

While intra-SMP latency (0.825 µs) is reasonable, intra-SMP bandwidth is poorer than expected: only 4,281 MiB/s out of an I/O channel bandwidth of 20 GB/s (19,072 MiB/s) [5]. Intra-SMP communication follows the same code path as intra-Cell communication (described in Section 4.2), which implies that the problem is with the Cell or QS21 hardware, not with the



(a) Latency as half the round-trip time

(b) Ping-pong bandwidth

Configuration	Latency (µs)	SPE clocks	Bandwidth (MiB/s)	SP flop/s per 4B word/s	DP flop/s per 8B word/s
Same Cell	0.272	870	22,944.2	4.3	0.6
Same SMP	0.825	2,640	4,281.3	22.8	3.3
Different SMPs	11.771	37,667	161.2	605.9	86.6

(c) Summary of point-to-point performance

Figure 5: Cell Messaging Layer point-to-point communication performance

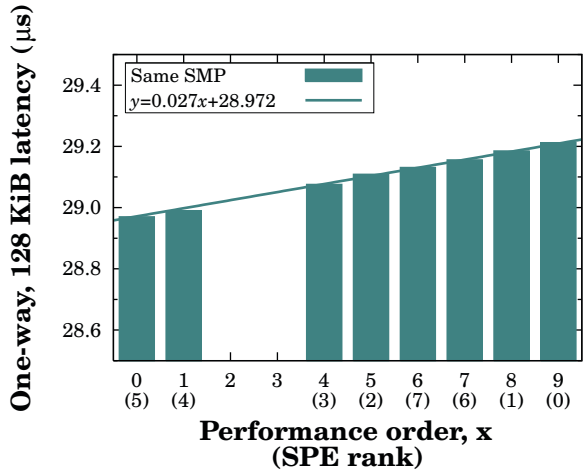
Cell Messaging Layer itself. To verify this hypothesis we performed two additional measurements, the results of which are shown in Figure 6. First, we analyzed the ½ RTT of a 128 KiB message between each of SPE ranks 0–7 (on the first Cell socket) and SPE rank 8 (on the second socket). As Figure 6(a) indicates, different SPEs observe different latencies to the BIF although there is no architecturally obvious explanation of the pattern. However, the difference from the best to worst performer is only 0.242 µs—not enough to explain the Cell Messaging Layer's low intra-SMP bandwidth.

Figure 6(b) varies the number of simultaneous 128 KiB ping-pongs between the two Cells (with rank P exchanging messages with rank $P+8$) and reports the aggregate bandwidth observed. The data show that an increase in simultaneous transfers implies an increase in aggregate bandwidth up to the asymptote at 20 GB/s. The conclusion is that the BIF multiplexes its offered bandwidth, possibly over the four rings of the EIB, implying that the Cell Messaging Layer's intra-SMP bandwidth of 4,281 MiB/s should be compared to 4,768 MiB/s (20 GB/s ÷ 4). That is, the Cell Messaging Layer's intra-SMP bandwidth is 90% of the channel's theoretical peak.

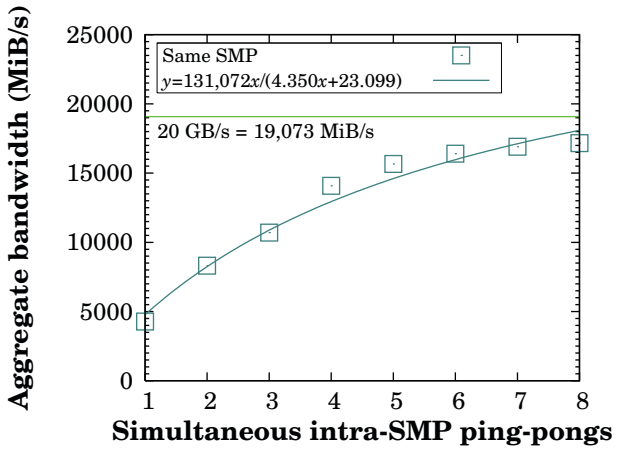
Node-to-node communication (11.7 µs latency and 161.2 MiB/s bandwidth) is slow—InfiniBand 4X's theoretical peak bandwidth is 953.7 MiB/s—but not representative. The QS21 boards on which we performed our experiments contain a hardware bug on the bridge to the PCIe I/O bus that affects correctness. The workaround put in place involves reducing the bridge's bandwidth to a quarter of its nominal value. Also, the 11.7 µs latency is only 2.9 µs higher than the PPE-to-PPE latency measured with an MPI latency benchmark. The conclusion is that the Cell Messaging Layer's SPE-PPE and PPE-SPE synchronization (Section 4.2) adds only 2.9 µs of latency to that of the underlying MPI implementation.

5.3. Comparative performance

To test our claim that receiver-initiated message passing offers an improvement in latency over sender-initiated message passing we compare the Cell Messaging Layer, which uses receiver-initiated message passing, to Kumar et al.'s Cell MPI implementation [17], which uses sender-initiated message passing. One challenge with such a comparison is that Kumar et al.'s MPI—henceforth to be called "buffered-mode MPI", after the title of their paper (although they in fact use synchronous-

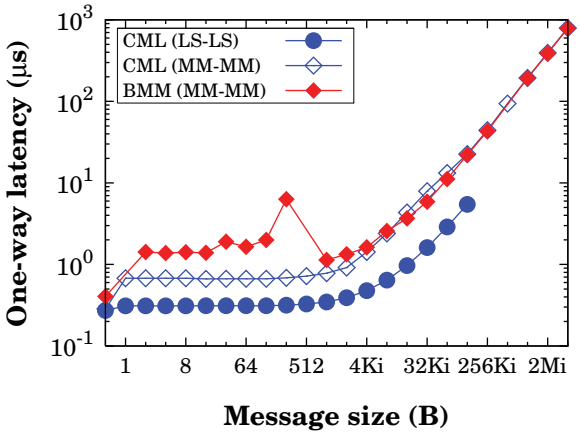


(a) Latency by rank

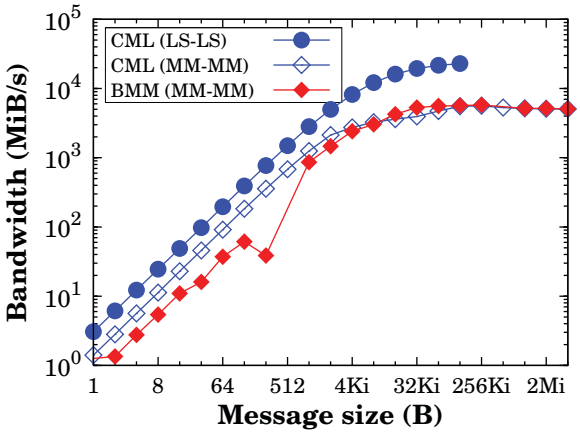


(b) Aggregate bandwidth

Figure 6: Diagnosis of low intra-SMP bandwidth



(a) Latency as half the round-trip time



(b) Ping-pong bandwidth

Figure 7: Cell Messaging Layer vs. buffered-mode MPI

mode communication for messages larger than 2 KiB)—transfers data between buffers in main memory while the Cell Messaging Layer transfers data between buffers in local store. To account for this different in approach, we wrote a few functions *on top of* the Cell Messaging Layer to transfer data between main-memory buffers. The sender repeatedly GETs 16 KiB blocks of data from main memory and `MPI_Send()`s them to the receiver, who `MPI_Receive()`s each block and PUTs it to main memory.

Figure 7 graphs the results of this now fair comparison. The "CML (LS–LS)" curves are the data from Figure 5 and are included for contrast. The "CML (MM–MM)" curves represent the performance of the main memory-to-main memory transfer. The "BMM (MM–MM)" curve is the buffered-mode MPI performance. Because the buffered-mode MPI implementation is not publicly available we extracted the performance data for the "BMM (MM–MM)" curve from Kumar et al.'s paper using the Engauge Digitizer.[2] Even with the extra control overhead layered above it, the Cell Messaging Layer observes a latency of 0.285 µs versus 0.41 µs for buffered-mode MPI—an improvement of 30.5%. While this represents an impressive speedup it is largely to be expected: receiver-initiated message passing requires only two network latencies versus sender-initiated message passing's three.

5.4. Collective-communication performance

We now briefly examine the performance of some of the Cell Messaging Layer's collective-communication operations. Figure 8 presents the latency of the Cell Messaging Layer's `MPI_Barrier()` and `MPI_Allreduce()` functions up to 64 SPEs (4 dual-Cell QS21 boards). As that figure shows,

[2] `http://digitizer.sourceforge.net/`

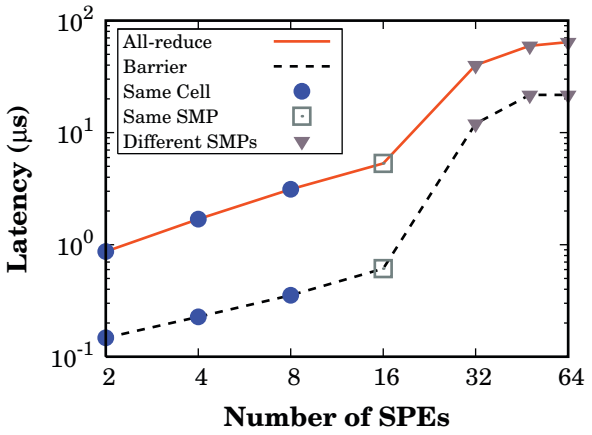


Figure 8: Latency of `MPI_Barrier()` and `MPI_Allreduce()`

the 8 SPEs within a Cell can be synchronized in 0.354 µs, just slightly more than a point-to-point latency. (Note that the Cell Messaging Layer uses a simpler protocol for barriers than for point-to-point operations.) `MPI_Allreduce()`, which is built atop `MPI_Send()` and `MPI_Recv()`, takes only 3.125 µs to sum one 32-bit integer per SPE and broadcast the result to all of the SPEs in a Cell. Performance is smooth across the SMP because the Cell Messaging Layer uses a binomial tree for the all-reduce, so a minimal number of messages crosses the slow inter-Cell link.

6. Conclusions

In this paper we argued for the use of *receiver-initiated message passing*, a rarely used technique for improving the latency of MPI-style communication atop RDMA networks. While the traditional implementation approach, sender-initiated message passing, has the benefit of being able to efficiently support wildcard receives (`MPI_ANY_SOURCE`), our argument is that on processors with high compute rates but small local memory capacity, such as the Cell Broadband Engine, it is more important to reduce communication latency than to provide the complete set of MPI communication semantics.

We implemented receiver-initiated message passing on the Cell and presented a set of performance measurements. Our data show a 30.5% improvement over the best reported latency for a Cell-based messaging layer discussed in the literature, one that uses sender-initiated message passing. We can therefore conclude that receiver-initiated message passing does offer a substantial performance benefit over sender-initiated message passing and is an implementation approach that should be given serious consideration for future messaging-layer implementations.

Acknowledgments

This work was supported by the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC52-06NA25396 with Los Alamos National Security, LLC. The author would like to acknowledge the Cell-related help and information—and suggestions for this paper—that he received from Greg Johnson, Mike Lang, Olaf Lubeck, Darren J. Kerbyson, Kei Davis, José Carlos Sancho-Pitarch, Adolfy Hoisie, and Kevin J. Barker from the LANL PAL team; and the information about MPI implementations he received from Brian Barrett and Galen Shipman of the LANL Open MPI team.

References

[1] Jon Beecroft, David Addison, Fabrizio Petrini, and Moray McLaren. Quadrics QsNet II: A network for supercomputing applications. In *Proceedings of Hot Chips 15*, Palo Alto, California, August 17–19, 2003.

[2] Matthias Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena R. Mesarina. Virtual-memory-mapped network interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[3] Stephan Brauss, Martin Frey, Anton Gunzinger, Martin Lienhard, and Josef Nemecek. Swiss-Tx communication libraries. In Peter M. A. Sloot, Marian Bubak, Alfons G. Hoekstra, and Louis O. Hertzberger, editors, *Proceedings of the 7th International Conference on High-Performance Computing and Networking (HPCN Europe 1999)*, volume 1593 of *Lecture Notes in Computer Science*, pages 623–632, Amsterdam, The Netherlands, April 12–14, 1999. Springer.

[4] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Computing*, 33(9):634–644, September 2007.

[5] Thomas Chen, Ram Raghavan, Jason N. Dale, and Eiji Iwata. Cell Broadband Engine Architecture and its first implementation—a performance view. *IBM Journal of Research and Development*, 51(5):559–572, September 2007.

[6] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract). *ACM SIGMETRICS Performance Evaluation Review*, 13(2):1–3, August 1985.

[7] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heteroge-*

neous Networks (HeteroPar'06), pages 1–9, Barcelona, Spain, September 25–28, 2006.

[8] Groupe de Recherche en Informatique Parallèle. Final CTI Swiss-Tx project report, June 2001.

[9] Jeff Hilland, Paul Culley, Jim Pinkerton, and Renato Recio. *RDMA Protocol Verbs Specification (Version 1.0)*. RDMA Consortium, April 2003.

[10] InfiniBand Trade Association. *InfiniBand Architecture Specification Release 1.2*, October 2004.

[11] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4(2):133–172, April 1987.

[12] Vijay Karamcheti and Andrew A. Chien. A comparison of architectural support for messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA 1995)*, pages 298–307, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM Press.

[13] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, May–June 2006.

[14] Kenneth R. Koch, Randal S. Baker, and Raymond E. Alcouffe. Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65(108):198–199, 1992.

[15] Murali Krishna, Arun Kumar, Naresh Jayam, Ganapathy Senthilkumar, Pallav Baruah, Raghunath Sharma, Shakti Kapoor, and Ashok Srinivasan. A synchronous mode MPI implementation on the Cell BE™ architecture. In Ivan Stojmenovic, Ruppa K. Thulasiram, Laurence T. Yang, Weijia Jia, Minyi Guo, and Rodrigo Fernandes de Mello, editors, *Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications (ISPA 2007)*, volume 4742 of *Lecture Notes in Computer Science*, pages 982–991, Niagara Falls, Canada, August 29–31, 2007. Springer.

[16] Arun Kumar, Naresh Jayam, Ganapathy Senthilkumar, Murali Krishna, Pallav K. Baruah, Ashok Srinivasan, Shakti Kapoor, and Raghunath Sharma. Exploration of the potential of Cell architecture for MPI applications. In *Poster Session, 2007 International Conference on High Performance Computing (HiPC 2007)*, Goa, India, December 18–21, 2007.

[17] Arun Kumar, Ganapathy Senthilkumar, Murali Krishna, Naresh Jayam, Pallav K. Baruah, Raghunath Sharma, Ashok Srinivasan, and Shakti Kapoor. A buffered-mode MPI implementation for the Cell BE™ processor. In Yong Shi, Geert Dick van Albada, Jack Dongarra, and Peter M.A. Sloot, editors, *Proceedings of the 7th International Conference on Computational Science (ICCS 2007), Part I*, volume 4487 of *Lecture Notes in Computer Science*, pages 603–610, Beijing, China, May 27–30, 2007. Springer.

[18] Myricom, Inc. GM: A message-passing system for Myrinet networks, 2.1.24. May 20, 2006.

[19] Ashwini K. Nanda, J. Randal Moulic, Robert E. Hanson, Gottfried Goldrian, Michael N. Day, Bruce D. D'Amora, and Sreeni Kesavarapu. Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. *IBM Journal of Research and Development*, 51(5):573–582, September 2007.

[20] Moriyoshi Ohara, Hiroshi Inoue, Yukihiko Sohda, Hideaki Komatsu, and Toshio Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 45(1):85–102, 2006.

[21] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (Supercomputing 95)*, San Diego, California, December 4–8, 1995. ACM/IEEE, ACM Press.

[22] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*, volume 1, The MPI Core. The MIT Press, Cambridge, Massachusetts, 2nd edition, September 1998.

[23] Technical Committee 25: Quantities and Units, and Their Letter Symbols. Letter symbols to be used in electrical technology—Part 2: Telecommunications and electronics. International Standard 60027-2, International Electrotechnical Commission, Geneva, Switzerland, November 2000.

[24] Don Towsley, Jim Kurose, and Sridhar Pingali. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. *IEEE Journal on Selected Areas in Communications*, 15(3):398–406, April 1997.

[25] Asimakis Tzamaloukas and J. J. Garcia-Luna-Aceves. A receiver-initiated collision-avoidance protocol for multi-channel networks. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 1, pages 189–198, Anchorage, Alaska, April 22–26, 2001. IEEE.

[26] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In Michael B. Jones, editor, *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 40–53, Copper Mountain, Colorado, December 3–6, 1995. ACM SIGOPS, ACM Press.

[27] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a mechanism for integrated communication and computation. In David Abramson and Jean-Luc Gaudiot, editors, *Proceedings of the 19th International Symposium on Computer Architecture (ISCA'92)*, pages 256–266, Gold Coast, Queensland, Australia, May 19–21, 1992. ACM Press.